Exploiting Parallelism by Customizing Evaluation

Silvia Clerici and Cristina Zoltan

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Barcelona, Spain

Abstract. NiMo is a totally graphic language from the family of Higher Order Typed languages with a strong Data flow inspiration. The interpreter is a specialized graph transformation system, and therefore the language operational semantics is given in terms of graph transformations. In NiMo parallelization is implicit and the evaluation policy is customizable following a process-centered approach. Here we explore some of the methodological possibilities that it opens. Some classical examples illustrate how combining modes greatly increases processor usage, decreases channel population, and achieves subnet synchronization in a very easy and intuitive way. We also present a stream programming technique and a real case application for generative and multistage-programming.

Keywords: Parallel Programming, Evaluation Policy, Data-Flow Languages, Functional Languages

1 Introduction

With the emergence of commodity multicore architectures, exploiting tightly-coupled parallelism has become increasingly important. Most of the parallelization efforts are addressed to applications that compute with large amounts of data in memory and in general have a regular behavior. In the scenario of tiny artifacts, interactive/reactive applications do not deal with huge amounts of data in memory, but streams of data. However they can still exploit several cores in a fine grain parallelization. Hence, finding simple ways to increase parallelism is now a matter of general and growing interest.

NiMo (Nets In Motion) is a graphic-functional-data flow language designed to visualize algorithms and their execution in an understandable way. The bi-dimensional representation displays the chain of process dependences revealing the implicit parallelism, and graphic execution helps us to understand where and when resources are used in the program, thus giving clues to optimize the solution. Programs are process networks that evolve showing their full state at each execution step. Processes are polymorphic, higher order and have multiple outputs. The language has a set of primitive processes well suited to stream programming and supports open programs and interactive debugging. The system provides an also graphic and incremental type inference system [1] that guarantees type-safeness by construction.

NiMo is a non-strict parallel language, but there are not explicit constructs to indicate parallelization. In the NiMo model all processes selected to act are supposed to execute in parallel at the same execution step. An execution step is a transition from one net to the next, where all the selected processes have produced the corresponding graph transformation. From the user point of view all of them have acted in parallel. The system provides facilities to measure the used resources (parallelism level, number of steps, number of processes, etc.).

The NiMo initial version followed a parallel lazy evaluation policy; nevertheless, the user could locally modify the evaluation order by setting an explicit requirement on any process. In the current version, a variety of modes determine the activeness level of each process. Modes can be globally or locally set for each process and dynamically changed giving the user a very intuitive and notably flexible way of customizing evaluation order. By tuning modes, the user can increase the number of processors that could act concurrently, regulate channel population or the evolution of the number of alive processes during execution. Furthermore, the evaluation modes could be used for deactivating subnets during experimentation or promoting speculative calculations, and to prevent evaluation of symbolic values. Since NiMo programs are graphs that evolve in execution, the operational semantics of the language was given in terms of graph transformation rules. It was presented in [2]. Here we discuss the possibilities opened by a flexible evaluation policy.

The next section gives a very brief introduction to the language constructs necessary to understand what follows. A more complete description of the language can be found in [3]. Section 3 summarizes the NiMo execution model, giving the repertory of process modes and a first example of their use. Section 4 presents a case study for the classical quicksort algorithm to illustrate how its performance can be drastically improved by only changing modes. Afterwards a further transformation increases even more the pipeline parallelism. Section 5 discusses the use of modes to handle symbolic execution and generative or multi-stage programming.

2 NiMo elements

NiMo programs are directed graphs with two kinds of nodes: processes and data items. Horizontal arrows represent channels of flowing data streams, and vertical arrows entering a process are non-channel parameters, which can also be processes. Processes can have any number of inputs and outputs, making the use of tuples unnecessary. There are neither patterns nor specific graphical syntax for conditionals. The main tokens are: rounded rectangles for processes, circles (or ovals) for constant values, black-dots for duplicators, and hexagons for data elements. Circles are labeled with their value for atomic types or with their names for symbolic constants of any type, even polymorphic. Hexagon labels are I, R, B, L and F for integers, reals, booleans, lists, and functional processes. Polymorphic data are labeled with "?". The NiMo syntax makes intensive use of color. In hexagons and circles it indicates their type, in process names it denotes the

evaluation mode, and edges have a state shown as a colored diamond to indicate process activation or data evaluation degree. All the mentioned nodes are interfaces having typed (in/out) connection ports. Interfaces are dragged from a ToolBox and dropped into the workspace where the net is being built. Clicking on a pair of ports connects them with an edge if both types are compatible; otherwise a failure message is generated. When a process output is connected the diamond is white, incoming data items are connected with a green diamond. The user can change the white diamond of a process output to red in order to request the process to act, or it can be changed in execution by its precedent process. There are two kind of processes: the gray ones are built-in processes (**bPs**) for basic types and stream processing, and the white ones are user defined processes (**nPs**). The repertory of **bPs** includes multiple output versions of many Haskell prelude functions, as the process *SplitCond*, that splits its entry channel according to the condition stated by the process connected at its vertical entry[1]. Also, some basic processes have configurable arity, as a *Map* with n input and m output channels (generalizing *map*, *zipWith* and *zipWith3*), *TakeWhile* and *Filter* with n input and output channels, and an *Apply* process. There are also two kind of hexagons: *list-item* hexagons for channel elements and *terminal hexagons* corresponding to the net outputs. Since the flow is from right to left, they are the leftmost interfaces. Subnets connected to a terminal hexagon are considered productive even when incomplete because in NiMo a process can execute even though its non-needed inputs are left open. In execution all the non-productive subnets are deleted by the garbage collector.

2.1 An Example

Fig. 1 shows the fifth execution step of a prime numbers generator (Eratosthenes

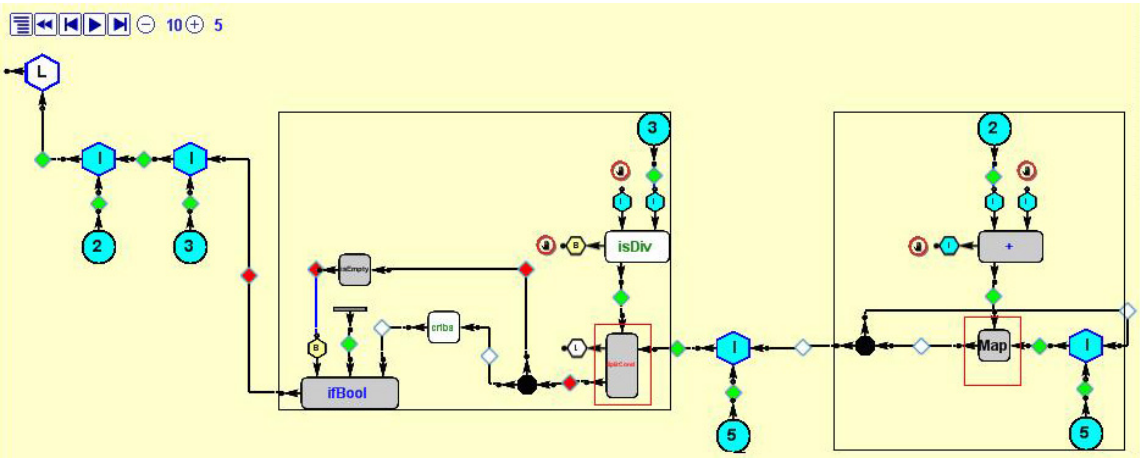

Fig. 1. Eratosthenes *sieve*, fifth step

sieve algorithm). In this case, the result (leftmost hexagon, labelled L) is a

[1] In Haskell code splitCond p l = (filter p l, filter notp l) where notp x= not(p x)

list with two already calculated elements (light blue hexagons labeled I) with constant values 2 and 3. The remaining elements are to be produced by the (recursive) subnet *sieve* whose entry channel at this step has a first element with value 5 and the next ones are to be produced by the subnet *fromStep* generating the odd numbers. The **bPs** *SplitCond* and *Map* are marked with a red frame. In the next step these will execute in parallel. The subnets *sieve* and *fromStep*

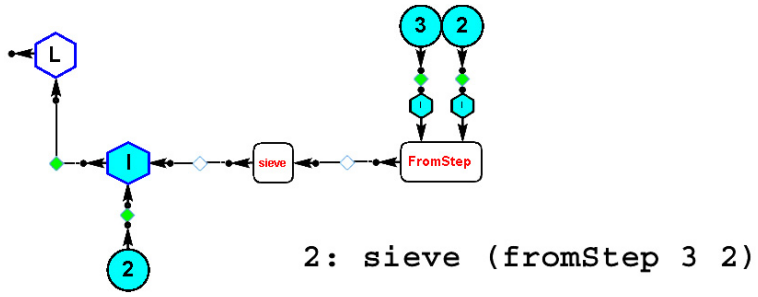

Fig. 2. Net *primes*

are net processes. Fig. 2 shows the initial net (and the equivalent Haskell code).

nPs are user-defined components whose interfaces (the white rounded-rectangles) are defined by means of a parameterization mechanism. The net in/out open ports that are to be considered as formal parameters or results are bound to the in/out ports of a configurable interface that is given a name. Later it can be imported to the Toolbox to be used as a process in a new net and so on, allowing incremental net complexity up to any arbitrary degree. At execution if the **nP** has to act, the white process interface is replaced by its associated net even if some of its parameters is not connected. Fig. 3 shows the net process definition

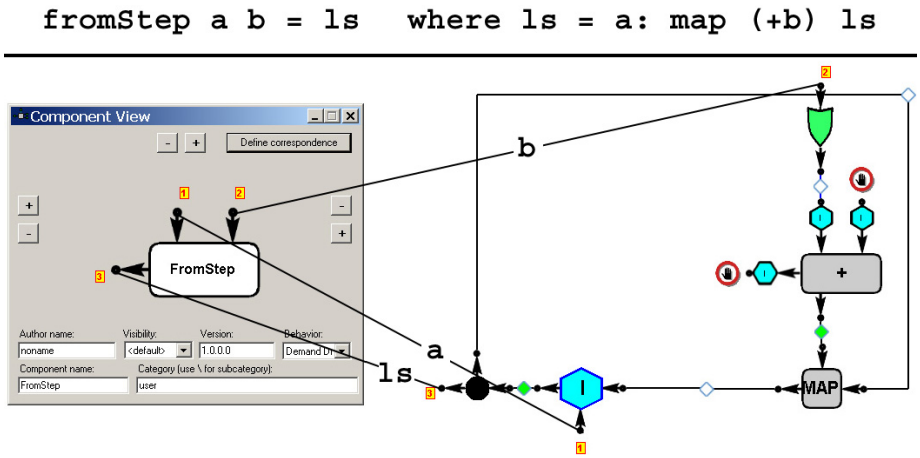

Fig. 3. Net process definition for *fromStep*

for *fromStep* and its correspondence with the equivalent Haskell code.

3 Executing nets. Process modes

In the initial version NiMo followed a parallel lazy policy. All processes acted only under demand (only when some of their outputs had a red diamond), except a distinguished one  that continuously forced its provider to act. The user could also set a demand on a given process by changing to red some of its output diamonds. A required process without enough data demanded the necessary input providers to act and so on (red diamonds propagation). Now the evaluation policy is no longer uniform. Several modes of increasing activeness can be set globally (for all) or locally for each process and can also be changed during execution. Modes alter the process scheduling without changing the code (only the color of process names changes). The modes for basic process are:

- **Disabled**: the process is not able to execute (even if requested).
- **Demand-Driven**: The process is able to execute only if requested.
- **End-Driven**: The process is also able to execute as soon as it can end and disappear (for instance, a non-required map can execute whenever any of its input channels ends).
- **Data-Driven**: The process is able to execute as soon as it has enough data.
- **Weak-Eager**: the process is always able to execute or to request its needed input providers.

Net processes have three possible modes:

- **Disabled**: will never expand;
- **Demand-Driven**: only when requested;
- **Auto-Expand**: always applies its expansion rule.

When only the outermost processes (the ones nearest to the net outputs) are set to Weak-Eager, and all the other processes are set to Demand-Driven the net has a total lazy parallel behavior. In order to simplify the net, Demand-Driven **bP** processes can be set to End-Driven. Setting all processes as Data-Driven gives the usual semantics in data flow approach. But an eager semantics cannot be emulated in NiMo by only changing modes (additional red diamonds must be set), because NiMo is a non-strict language and Weak-Eager processes require only their needed inputs. The process-centered approach makes possible to have the advantages and overcome the drawbacks of all the standard evaluation policies. The following section shows how an appropriate combination of modes allows increasing the implicit parallelism, dealing with subnets synchronization and regulating channel population. The example illustrates the use of the system tools for measuring the execution behavior.

3.1 Experimenting with modes

Lets consider again the prime numbers example in section 2.1. The net definition of *sieve* is shown in Fig. 4. A roughly equivalent Haskell code (since in NiMo there are not patterns nor special syntax for conditionals) is the following:

```
sieve [] = []
sieve (a:x) = a : sieve (filter (nodiv a) x)
nodiv a n = ( mod n a) > 0
```

The *sieve*'s input list is the entry of its right-most process *Hd-Tl*. The head of the entry list is duplicated to be the first generated output of *sieve* and the (second order) parameter of *isDiv*. Process *SplitCond* splits the tail of the entry

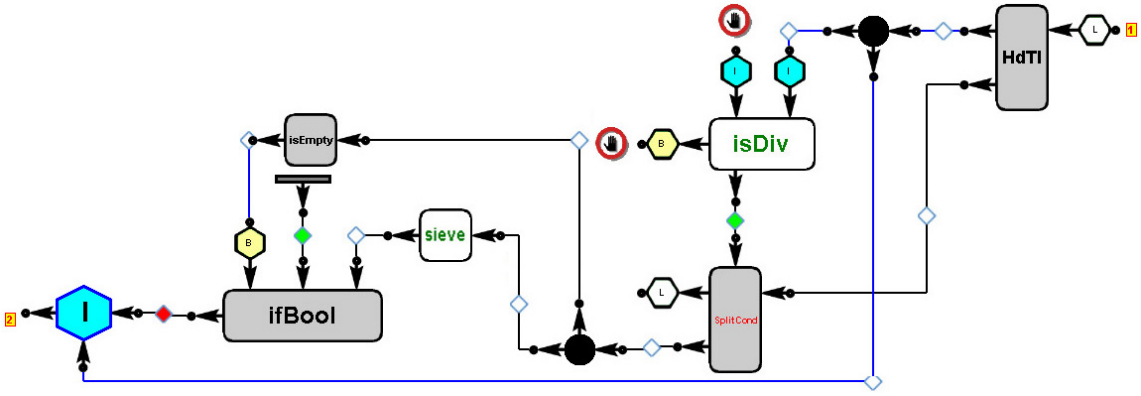

Fig. 4. Net *sieve*

list into two channels, but the first (with those that satisfy the condition) is left open, then acting as a filter for the opposite condition. The resulting values are the entry for the recursive call of *sieve*, once proven the list is not empty by means of the conditional process *ifBool*. In fact one could argue that in this context it will never happen and we could eliminate the conditional process. That is true, but in NiMo a net process interface (not in Disabled mode) is replaced by its internal net definition whenever it is required to act, i.e. when its has a red diamond in any of its outputs, which persists in the bound internal process output. Hence, even not having yet a first element to act the *sieve* net would be expanded, then the same would occur with its internal *sieve* interface and so on, with the unnecessary waste of memory and screen. The conditional overcomes this problem[2] because its condition only can be evaluated when a new value is produced by *SplitCond*. At this time the process *ifBool* disappears (together with its first and second parameters), *sieve* interface takes its place maintaining the red diamond in its output connection and it expands in the next step.

Hereafter we analyze how process modes impact the network performance. In the first experiment we choose a lazy policy, i.e. all processes (in the first level net and also in the subnets) are set to demand driven mode and an activator process  is added to force a continuous demand in the *primes* net output. Fig. 5 shows the execution at the time when the prime 11 is generated. According to the system step counter this take place at step 89 and we can see two system resource viewers. The one on top shows that most of the time there is only

[2] As happens with the Haskell patterns.

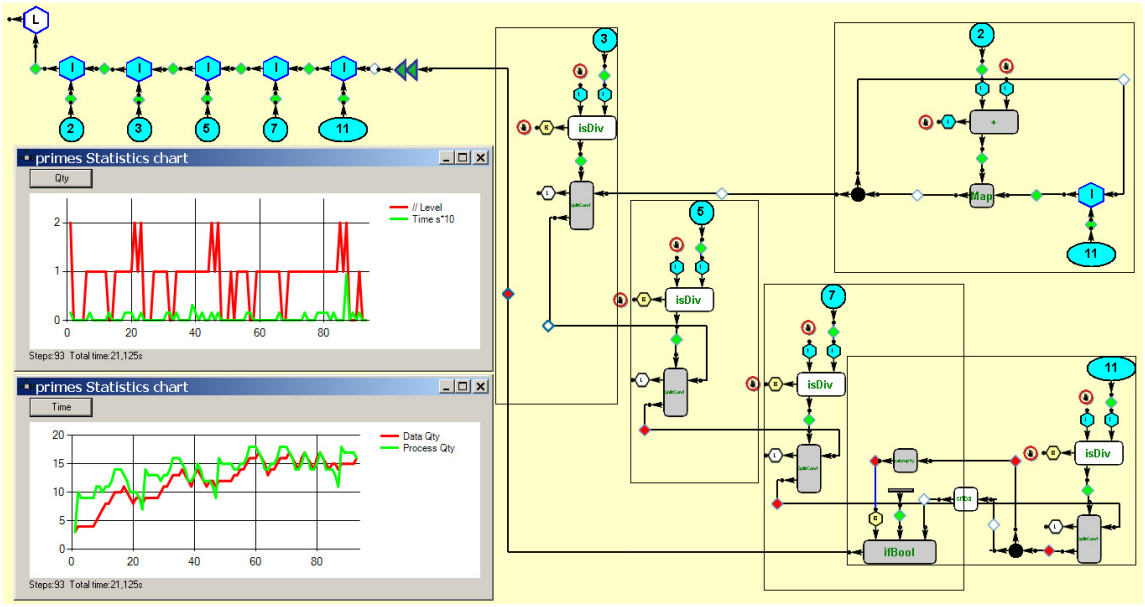

Fig. 5. All processes Demand-Driven

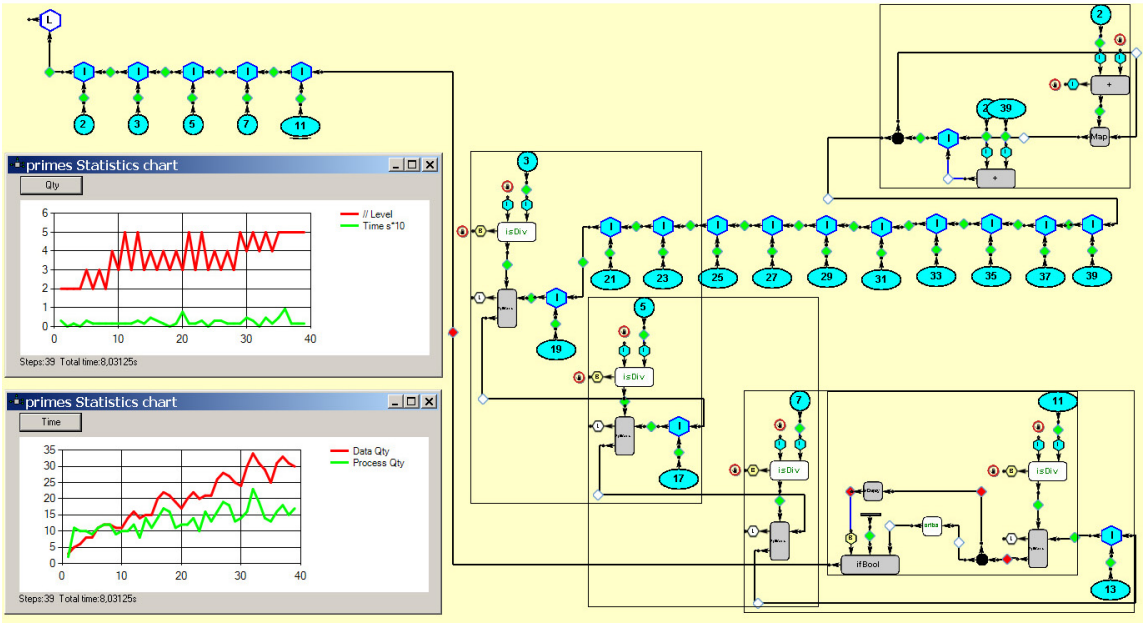

Fig. 6. All processes Data-Driven

one process acting, except when *fromStep* also acts in parallel to produce the next odd number. The other one shows that the number of data and processes are almost the same. A lazy approach in general produces low parallelism but maintains the channels size bounded.

In the second try all (**bPs**) processes are set to Data-Driven. Fig. 6 also shows the execution at the time when the prime 11 is generated. In this case it is produced at step 39, and in the resource viewer on the top it can be seen that the parallelism level increased, being bounded by the number of already

generated primes. Also, we can see that the number of elements in the channel connecting subnets *fromStep* and *sieve* increases as well (let observe the graphic of process/data). At this time there are ten already generated odd numbers waiting to be treated. It happens because the productive rhythm of *fromStep* is much higher than the consumption capacity of *sieve*. In general, the Data-Driven approach, common in data-flow languages, increases the parallelism since processes act as soon as they get their data, but this behavior can saturate channels.

The way of keeping the advantages of both policies is to use a lazy approach only in the places where the consumer is not able to process at the same rate of its producer. In this case, changing the duplicator's mode to Demand-Driven since it is the leftmost process of *fromStep*. But this implies that the consumer process in *sieve* has to be changed to Weak-Eager in order to activate the duplicator once having processed the previous element. The leftmost process in *sieve* is *Hd-Tl* but it dies after the first computation step, therefore the process that must keep the demand on *fromStep* is *SplitCond*. With just these changes we get a nearly equivalent amount of parallelism, since 11 is obtained in step 41 without overpopulating the channel.

4 Example: QuickSort NiMo Program

In this section we use a simple quicksort to show how modes for processes are determined. Due to experimentation, a new algorithm is presented along with the notion of *active stream*.

As remarked in [4], Quicksort remains one of the most studied algorithms in computer science. The version therein presented exploits concurrency without locking or explicit synchronization. The goal is obtained by using mutable store in a RAM memory. The algorithm is presented in Orc [5], which is a language based on a calculus with very few combinators; one of them is the parallel combinator. As we already said, in NiMo there is no explicit construct for signaling possible parallel process execution; processes can be executed in parallel if they are able to act[3] in the same computation step.

The classical simple QuickSort algorithm in Haskell code is

```
qsort []     = []
qsort (x:xs) = qsort less ++ [x] ++ qsort more
     where
          less = filter (<x)  xs
          more = filter (>=x) xs
```

Fig. 7 shows the equivalent NiMo net process definition where the conditional process *ifbool* and the access process *Hd-Tl* replace the pattern matching mechanism not present in NiMo. Reading the component from left to right, the concatenation process (*++*) brings together the elements that are smaller than

[3] have all the required values and their modes allow them to act.

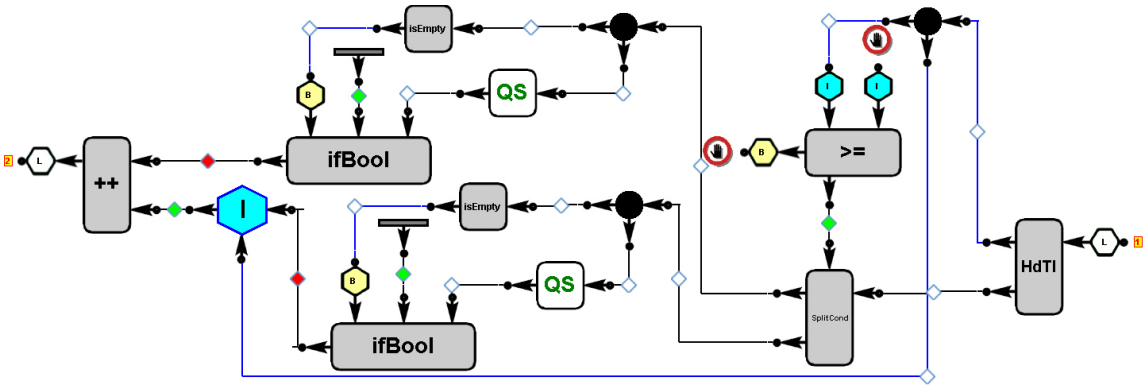

Fig. 7. QuickSort net

the pivot (if any) and the elements greater than or equal to the pivot. The input's head is duplicated to become the pivot and also the second order parameter of *SplitCond* that splits the tail of the entry into the two segments. Each one is the entry for two recursive applications of *QS*, once proven they are not empty. If not, an empty list is produced as the result. If we set all the processes in Fig.

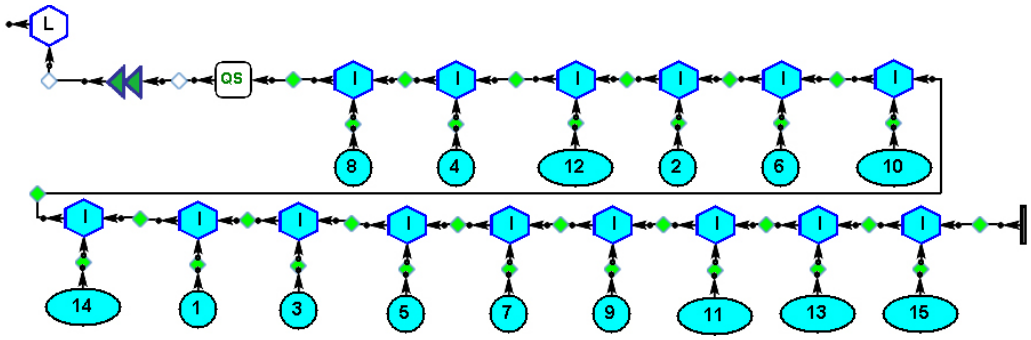

Fig. 8. NiMo program to test *QS* with processes in Demand-Driven mode

7 to Demand-Driven the net will have a lazy behavior. Hence, the program in Fig. 8 that uses *QS* has a process to force a continuous demand on the net. The entry list is a sample of 15 values in the best case, i.e values in a balanced tree shape. The execution of this version takes 303 steps.

If we set all **bPs** in the net *QS* to Data-Driven, leaving the internal *QS* processes as Demand-Driven, and change to red the diamonds in front of both *ifBool*, red diamonds will activate *QS* only if they need to be expanded. To run this modified version of *QS*, the activator process in Fig. 8 can be eliminated and the *QS* process must be set to Auto-Expand. By changing these modes, the number of steps for the same input is 61 instead of 303.

This version of *QS* exploits the implicit parallelism induced by the divide and conquer strategy. Looking at the algorithm execution, we can see that the process (*++*) treats its first operand element by element. All the values in the

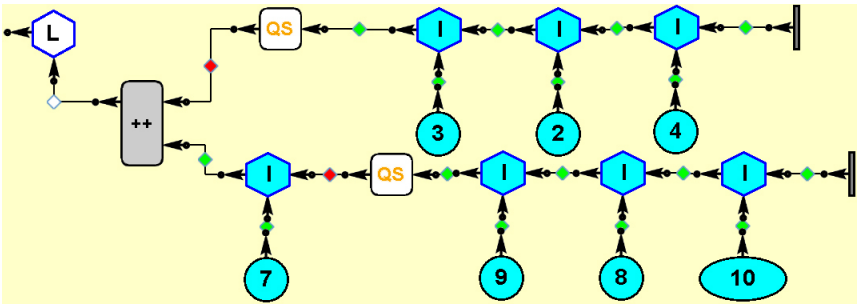

Fig. 9. Setting *QS* as Disabled

first operand need to be produced before the process can glue the second operand to construct the result (this operation takes one execution step). In Fig. 9 we have the result of executing a single recursive call to *QS* with the sequence 7,3,9,2,4,8,10 which corresponds to a balanced binary tree[4]. The initial pivot is 7. For obtaining this picture we have executed the program setting as Disabled the *QS* processes in the *QS* definition, in order to block the activation of recursive calls.

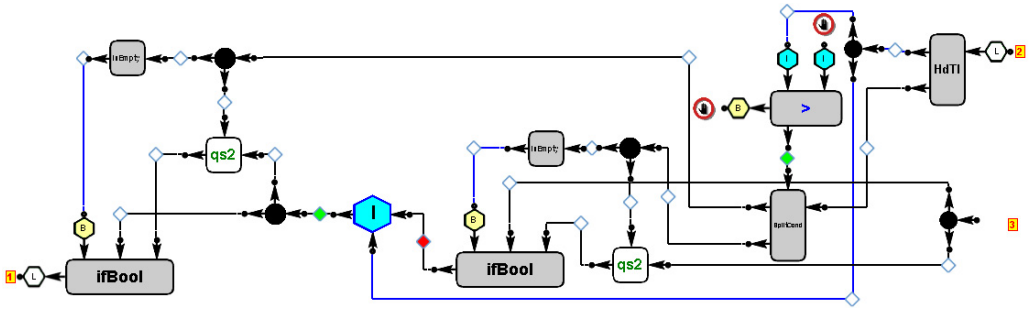

Fig. 10. Sorting without concatenation

We can eliminate the delay produced by the concatenation by transforming the net in what we call an active stream. Instead of constructing (using the concatenate process) the output stream, the net associated to the sorting procedure can be seen as a stream of processes and values i.e. an active stream. The definition for the new net process (*QS2*) is the one on Fig. 10.

Processes *QS2* have two input parameters: the stream to be sorted (top right) and another input parameter which is the rest of the stream (bottom right). The process *QS2* will be replaced by the ordered permutation of its first parameter and glued to its second parameter. The result is the sequence formed by (*QS2*) applied to the stream of values smaller than the pivot, followed by the pivot followed by another (*QS2*) having as its first parameter the stream of values greater or equal to the pivot and as its second the rest of the stream.

[4] The tree [7,[3,[2],[4]],[9,[8],[10]]]

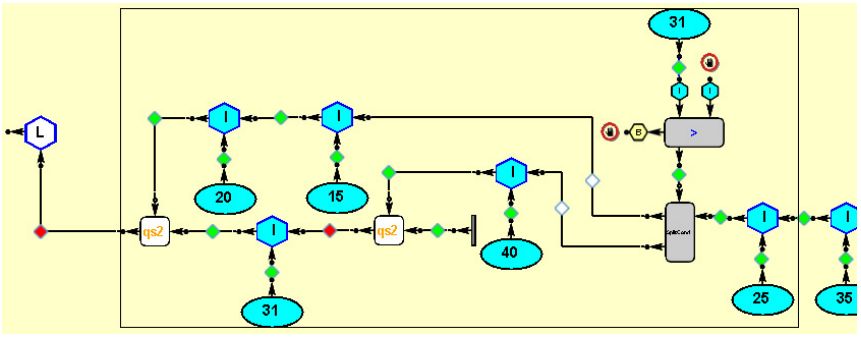

Fig. 11. Splitting the input with pivot 31

In Fig. 11 we can see the sequence construction.

We compare the behavior of the three algorithms using sample inputs like a balanced binary tree, which is its best behavior, the sequence of 15 elements in strict descending order([15..1]) and the sequence already sorted ([1..15]).

Table 1. Comparing three quick sort

Method	Input	Steps	steps for first
QS-lazy	[1..15]	496	52
QS	[1..15]	119	49
QS2	[1..15]	119	48
QS-lazy	[15..1]	678	528
QS	[15..1]	146	133
QS2	[15..1]	121	121
QS-lazy	balanc tree	303	110
QS	balanc tree	61	55
QS2	balanc tree	60	51

Measures used in Table 1 are the total number of steps needed to sort the sequences, and the number of steps needed for the result to start flowing. In the case where the input sequence is ordered there is almost no difference between *QS* and *QS2* because there is almost no delay motivated by concatenation. We see that when the input is the descending sequence (the worst case for quicksort method), the output for *QS2* starts flowing much earlier and takes almost as long as sorting the entire sequence[5].

5 Symbolic Computation and Generative Programming

The possibility of disabling processes has multiple applications. When the process is not yet defined it is the way to prevent it from being forced to act. In

[5] This algorithm runs faster and even uses less space than the classical version also in Haskell.

addition, set to Disabled some processes allows partial testing of subnets, or executing the algorithm by levels enabling to reason about the net structure in the intermediate states. This section discusses other two additional uses of disabling processes for symbolic computation and generative programming.

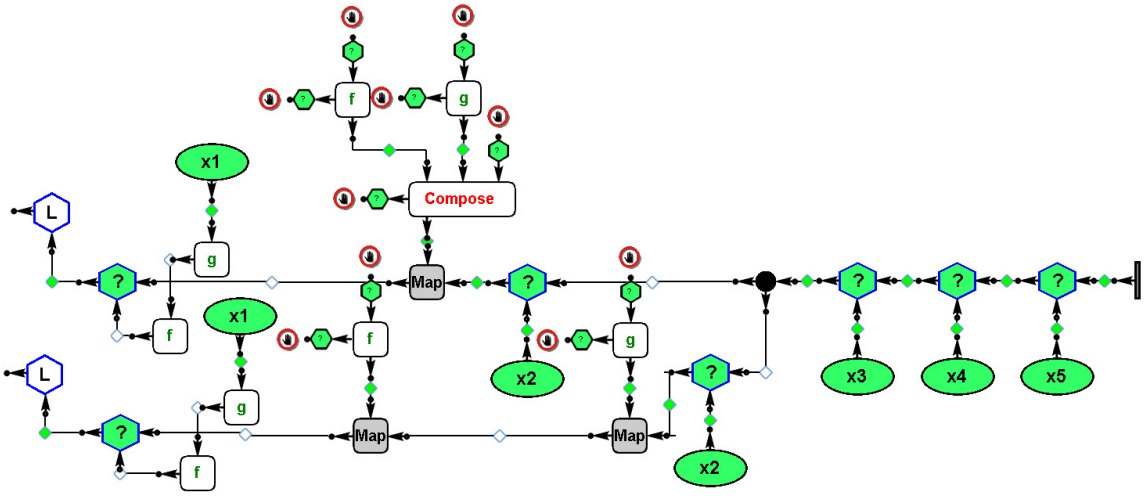

Fig. 12. Symbolic execution

The first one is here illustrated with an example where equivalent codes are compared in an abstract way. In NiMo there are symbolic constants of any type, even polymorphic. Fig. 12 shows the initial results of evaluating $map(f \circ g)[x1, x2, x3, x4, x5]$ and $map\ f(map\ g[x1, x2, x3, x4, x5])$. In this case the Disabled processes f and g are being used as symbolic functional values; they do not have an associated net process definition.

Regarding generative programming the idea is to define a net that generates a second one with some Disabled processes. The resulting net is the desired program. In order to be executed it is only required to globally set the Disabled processes to another mode, and execution proceeds in the next step (multi stage execution) or else the resulting net can be stored to be run later. This technique has been used (in collaboration with the UPC team of the WISEBED project [6]) to generate different topologies for sensor networks of variable size and afterward simulating their behavior [7].

The example in Fig. 13 corresponds to the classical problem of calculating average temperature using meteorological sensors. The initial net (on the left) is parameterized by the topology (ring, tree, star) and number of sensors. Its execution generates a network with the desired topology and all sensors having Disabled mode so that no one executes until the network is completely constructed since all of them have to produce the average temperature at the same time. Each sensor process is parameterized by the delay to produce its first output according to its position. It is simulated by means of a chain of activator processes (taking one execution step each). Generation also includes processes that simulate the input from the environment for each sensor. The net on the

right of Fig. 13 shows the fifth step of the generation stage. The initial net evolves

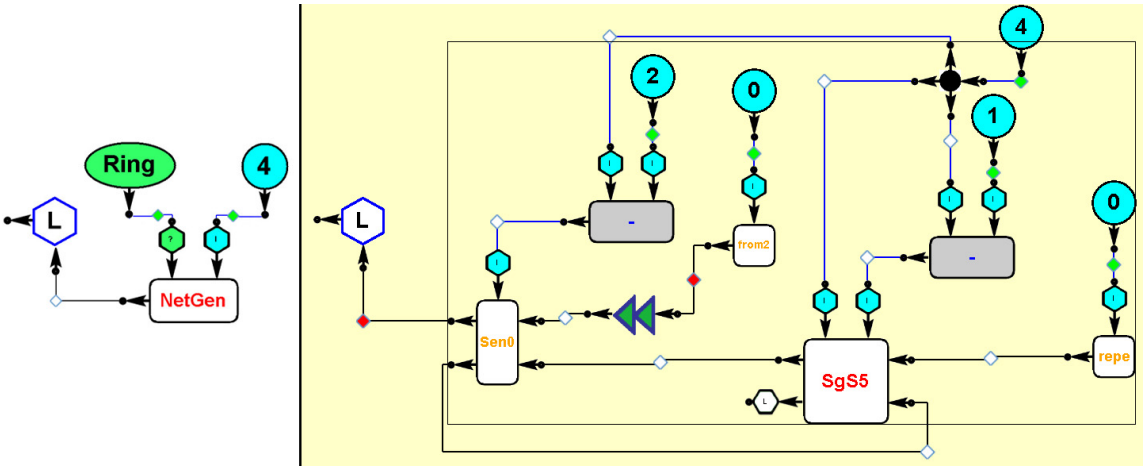

Fig. 13. Generative programming

until no more processes can act. Now by only changing globally the modes in Fig. 14 the execution becomes the sensor network simulation. Fig. 15 shows the

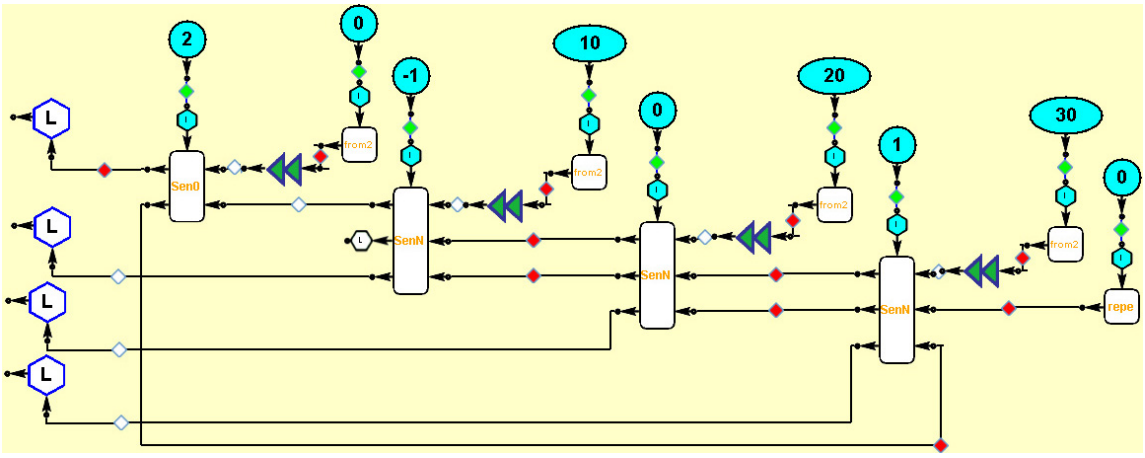

Fig. 14. Four sensors in ring topology

net state when two values were produced in each sensor output. Fig. 16 shows the generation stage result when the topology is a binary tree with depth 3.

Another example of multistage execution, in this case combining symbolic and numeric computation, is the generation of the Newton's binomial expansion for integer constants with symbolic values a and b and numeric value for the exponent. The arithmetic operator processes ($+$, $*$ and $\hat{}$) with symbolic parameters are set to Disabled mode. The execution stops once generated the binomial

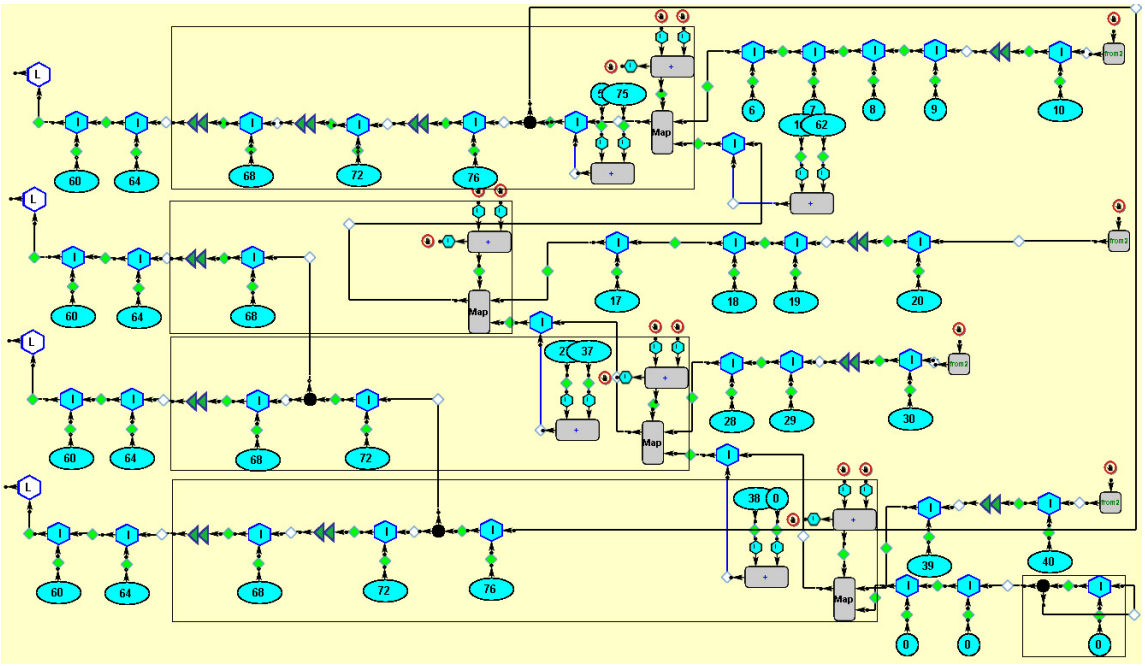

Fig. 15. Running the simulation

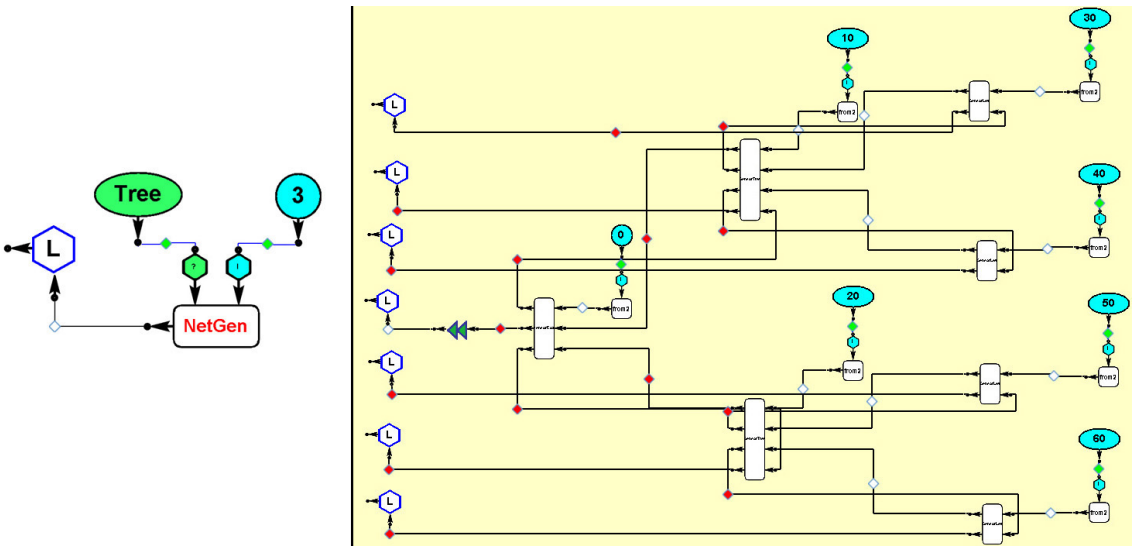

Fig. 16. Binary tree topology with 3 levels

expansion. If we run the same net with numeric values instead of a and b[6], once generated the expansion we can change to Data-Driven the Disabled mode for all the processes in the resulting net, the execution continues and the numeric final value is produced.

[6] Or if we replace all occurrences of values a and b in the expansion.

6 Conclusion

In programming languages exploiting parallelism, there are constructs or annotations that the programmer uses to parallelize or help the compiler to do so. The presentation of the program changes with these additions. With this concern in mind, [8] introduces *strategies*. In this approach "A function definition is split in two parts, the algorithm and the *strategy*, with values defined in the former being manipulated in the latter". *Strategies* allow establishing individual evaluation degrees for elements present in function definitions. They are associated to data types and for instance can describe how deeply individual elements of a list are to be evaluated. *Strategies* are used also for controlling parallelism.

In NiMo parallelism is given by the net structure. Different branches can be executed in parallel, different parameters of a process can be evaluated in parallel. Pipeline parallelism occurs when in a chain of processes several ones acquire enough data to act. The NiMo programmer strategy focuses on determining the activeness degree for each process. Also, as the annotations are just the color on the names of processes or on diamonds, the structure of the program remains exactly the same keeping the mental model. The language minimizes the need for programmer involvement in identifying parallelization. Using the classification given in [9] we can say that in NiMo the programmer could design its algorithms being aware that the execution will be done in a parallel model of execution (second level of abstractness), but the algorithm also runs and is efficient in a single processor model of computation. The first goal is to obtain a correct solution. Users have total control over their code and the state of its execution, they can interactively modify any program element undo steps, and store any state as a new net, therefore the system acts as an online tracer and debugger. On this regard, Disabled mode allows testing pieces of nets or visualizing the algorithm structure in intermediate states. Afterwards, the program can be tuned by observing the net behavior and the resource system viewers in order to accelerate execution and/or reduce memory usage.

On the other hand, in NiMo processes with unbound or symbolic parameters can execute. Disabling processes is also the way of integrating symbolic computation in the same framework and handling incompleteness. It also provides the means for generative programming and multi-stage execution.

The term "generative" in software development is associated to a system-family approach, which focuses on automating the creation of system-family members[10]. A variant is multi staged programming [11] which makes simplifications of generic programs developed using good abstraction mechanisms. This improves the efficiency of the resulting programs. Both approaches need language extensions (meta-programming or Domain Specific Languages).

In our context the term refers to having a generating net whose final result is a new executable net. If the original net is parameterized then there is a family of resulting nets. In general, the existence of Disabled processes may stop the

execution, which can be resumed by only changing their modes.

As a final conclusion we can say that by setting modes the user can modify the parallel scheduling in a simple and intuitive way since:

- modes define several levels of increasing activity (vs. the usual dichotomy lazy/eager).
- the process-centered approach allows detailed customizable evaluation
- modes can be dynamically changed global or locally

These characteristics give a notable flexibility in program tuning for resource usage optimization.

References

1. Clerici, S., Zoltan, C., Prestigiacomo, G.: Graphical type inference. a graph grammar definition. In Rex Page, Zoltán Horváth, V.Z., ed.: Revised Selected Papers from the Eleventh Symposium on Trends in Functional Programming, TFP 2010, Norman, OK, USA, 17-19 May 2010. Volume 6546 of LNCS., Springer (2011)
2. Clerici, S., Zoltan, C.: A dynamically customizable process-centered evaluation model. In: PPDP '09: Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming, New York, NY, USA, ACM (2009) 37–48
3. Clerici, S., Zoltan, C., Prestigiacomo, G.: Nimotoons: a totally graphic workbench for program tuning and experimentation. Electronic Notes in Theoretical Computer Science **258**(1) (2009) 93 – 107 Proceedings of the Ninth Spanish Conference on Programming and Languages (PROLE 2009).
4. Kitchin, D., Quark, A., Misra, J.: Quicksort: Combining concurrency, recursion, and mutable data structures. In Roscoe, A., Jones, C.B., Wood, K.R., eds.: Reflections on the Work of C.A.R. Hoare. History of Computing. Springer London (2010) 229–254 10.1007/978-1-84882-912-1_11.
5. Kitchin, D., Quark, A., Cook, W., Misra, J.: The orc programming language. In: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems. FMOODS '09/FORTE '09, Berlin, Heidelberg, Springer-Verlag (2009) 1–25
6. : Wisebed. `http://www.wisebed.eu` (2010)
7. Clerici, S., Duch, A., Zoltan, C.: Implementing static synchronus sensor fields using nimo. `http://www.lsi.upc.edu/dept/techreps/` (2009)
8. Trinder, P.W., Hammond, K., L, W.S., Jones, P.: Algorithm + strategy = parallelism. Journal of Functional Programming **8** (1998) 23–60
9. Skillicorn, D.B., Talia, D.: Models and languages for parallel computation. ACM Computing Surveys **30** (1996) 123–169
10. Czarnecki, K. In: Overview of Generative Software Development. Volume 3566 of Lecuture Notes in Computer Science. Springer-Verlag (2004) 326–341
11. Taha, W.: Generative and transformational techniques in software engineering ii. Springer-Verlag, Berlin, Heidelberg (2008) 260–290